# Engineering Degrees of Agency

Steven P. Fonseca
QSS Group Inc., NASA Ames
MS 269-2
Moffett Field, CA 94043
011 650 604 1083
fonseca@email.arc.nasa.gov

## ABSTRACT

The Mission Control Technologies Project at NASA Ames Research Center is developing component-based middleware with multi-agent like characteristics that must satisfy many competing quality attributes. This paper makes the observation that, while a multi-agent system solution is a relevant source of architecture and design artifacts, it is not possible to achieve the desired system quality attributes with a purely MAS implementation. MAS frameworks offer agents as the primary unit of decomposition and encapsulation. The degree of agency is also selected by framework developers – indicating that the agent abstraction is not considered a point of framework variability. We introduce the notion of engineering degrees of agency into an application framework by designing points of variability (hooks) that enable a programmer to tune the degree of agency used through customizations of the agent abstraction.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, frameworks, patterns, modules.*

## General Terms

Design, Reliability, Languages.

## Keywords

Multi-agent system, framework, component, agent-oriented software engineering, semantic web, reuse, distributed architecture.

## 1. C3I REQUIREMENTS AND DESIGN INFLUENCES

Operations software at NASA has traditionally been written to satisfy narrow sets of requirements that are driven by specific mission needs. Within the bounds of supporting a single mission, the software infrastructure used is a largely non-integrated collection of applications. The result of this approach to enterprise information system (EIS) design is a severely limited ability to achieve reasonable levels of interoperability, flexibility, evolvability, reliability, and other quality attributes that make building these systems cost efficient and ensure that the system performs as required. As with many organizations faced with satisfying similar EIS requirements, achieving such quality attributes are of primary importance to the Control, Communication and Information (C3I) architecture under development at NASA as part of the Constellation Project for continuing human exploration of space. Constellation envisions missions where the number of elements and the number of their interactions has significantly increased in comparison to today's missions. Furthermore, and in stark contrast to the closed system design of past and current missions, it is predicted that new kinds of elements will be routinely introduced to the system throughout the lifetime of a mission. In the abstract, similar capabilities are afforded by the agent-oriented programming paradigm (herein refereed to as AOP, not to be confused with the more common acronym used for aspect-oriented programming).

The hallmark of the agent-oriented programming paradigm is the decomposition of a system into loosely coupled and autonomous software agents that readily interact with other software agents and their environment. Although multi-agent systems (MAS) can be closed, this programming paradigm guides designs toward being open and offering a flavor of service-oriented architecture (SOA). As with SOA, but in the vernacular and context of the C3I, "Applications will be modular and support a decoupling of 'what' the functionality provides from 'how' the functionality is provided. Application component reuse and sharing of common capabilities between C3I Instances…will be required to reduce the cost and solve the temporal restrictions imposed by deep space missions" [10]. This statement is from one of a number of documents coming online as the conceptual framework and requirements for C3I are developed. These documents indicate that, to a first approximation, agent-oriented programming and multi-agent system design abstractions can be leveraged and are a natural fit for realizing the C3I's envisioned system. The key system characteristics desired by C3I that are supported by multi-agent systems solutions include:

- Independent software elements
- Highly interactive software elements
- Element interaction via loosely coupled message exchange
- Message level connectivity standards
- Semantic interoperability achieved by information models (ontologies)
- Software elements are consumers of declaratively specified knowledge
- The system exhibits intelligence including the ability for inference, with the particular long term vision of having an adaptable, self-configuring system

## 1.1 C3I Quality Attributes

The characteristics described above establish the relevance of the agent-oriented programming paradigm to the design of the C3I system but do not provide an indication of just how well they satisfy the quality attributes of interoperability, flexibility, evolvability, and reliability. As is known [1] and can be intuitively predicted, the achievement of quality attributes requires designers to balance system capabilities since attributes are frequently dependent on one another and can share inverse relationships. The question then becomes to what degree the balance of quality attributes of a typical multi-agent system solution match-up with that most appropriate for the C3I system. The follow-on question and a focus of this paper is how can the agent-oriented programming paradigm be adapted and augmented to satisfy the C3I system requirements.

Beginning first with interoperability, a high level assessment is provided of the shortcomings of using the AOP paradigm and technology bases to satisfy several of the C3I quality attributes. Abstractions supporting interoperability between agents are well-supported by AOP, whose major contributions are the development of agent communication languages and grounding the semantics of messages through the use of ontologies (originated by the artificial intelligence community). However, to date the widespread industrial adoption to AOP originating communication languages has yet to take hold, with the open distributed systems community at large instead embracing web services technologies and their successor the semantic web technology base. An additional observation is that AOP systems are typically implemented with academic and industrial research lab frameworks. Across such platforms, interoperability can only occur at the granularity of an agent but it is very common for such agents to be further decomposed into potentially reusable behaviors. These behaviors are typically not executable outside of their local context. In summary, the C3I interoperability requirement is better met by adherence to standards that are more widespread and where the more fine-grained behaviors of the system, while possibly strategically hidden, can also be readily exposed to and consumed by any software element in the system.

The AOP paradigm and its instantiations provide high levels of flexibility through agent autonomy of action, dynamism of agent presence, loose coupling of communication connectivity, and autonomy of agent behavior change. This level of flexibility is too high for C3I and must be balanced with the desire to achieve reasonable levels of system verifiability and reliability. Central to the notion of autonomous agents is its ability to control when and how it acts. While this level of flexibility may be appropriate for some portions of the C3I application space, there are certainly many mission-critical functions where it is essential to have software execute in accordance with contractually specified interfaces of some kind. Dynamism of agent presence where agents can be introduced and removed dynamically to a running multi-agent system is in principle well-aligned with this facet of C3I flexibility. Loose coupling of communication connectivity is also well-aligned with C3I needs but pragmatic engineering considerations make unlikely the direct reuse of most of the standard AOP design idioms. As a particular example, messages exchanged between agents are typically accompanied with significant interoperability-facilitating metadata. This is a noteworthy expense when the actual message is short or the agents already share enough knowledge to communicate, not to mention that the application programmer is burdened with having to code-in the metadata. Finally, AOP and its instantiations permit the dynamic additional and removal of agent behavior that is communicated via publication and cancellation of agent services. While it may be appropriate for some portions of the C3I application space to be this flexible, greater control over behavior availability must exist to increase system reliability.

Agent-level evolvability is typically provided by multi-agent system frameworks that support the introduction of new agents to the system where these agents typically are managed in accordance with a lifecycle model [3]. This runtime mechanism provides a starting point for achieving C3I system evolvability. Two abilities that are not offered by typical MAS framework are agent versioning where updates can be managed in fine-grain detail and an infrastructure that propagates updates to copies of software agents. In AOP, only one such agent with a unique identity exists in the system but the component-based system of C3I envisions replicated instances that would require a code synchronization algorithm and infra-structure support for component updates. Loose coupling of software elements is an essential requirement for the system to evolve but was already addressed in the context of system flexibility.

Quality attributes are an important consideration in developing the C3I design but is not the only one. Functional capabilities and the leveragability of third party software also influence design. These kinds of considerations are periodically identified as part of the design rationale throughout this paper.

## 1.2 Balancing Design Tradeoffs

From the discussion above it can be seen that the AOP paradigm and its instantiations can be leveraged in designing the C3I architecture. It is also clear that both adaptations to AOP are required and that design artifacts from other distributed system approaches are preferred in some cases. Considering adaptations to AOP, one makes the observation that MAS solutions provide distinctly AOP framework supporting instantiations. MAS frameworks offer agents as the primary unit of decomposition and encapsulation. The degree of agency is also selected by framework developers – indicating that the agent abstraction is not considered a point of framework variability. The effect of this common thinking within the community of developers is MAS framework implementations that support their own specific flavor of agency (generally emphasizing characteristics that are inline with an anticipated application domain or research interest) that cannot be readily adapted. Analyzing the C3I system requirements leads to the hypothesis that to successfully deploy large-scale multi-agent systems requires frameworks that permit programmers to choose the degree and character of agency used for a particular application implementation context. This flexibility is critical in developing enterprise information system architectures, such as that envisioned by the C3I, to optimally balance a complexly related set of quality attributes. The goal in engineering degrees of agency is to build into an application framework the points of variability (hooks) that enable a programmer to tune the degree of agency used through customizations of the agent abstraction.

## 2. MCT COMPONENT MODEL

Mission Control Technologies is a project within the Intelligent Systems Division at NASA Ames Research Center that is a contributing member of the C3I effort. MCT endeavors to create a component-based infrastructure that supports mission operations where its set of frameworks provides a comprehensive suite of system capabilities including distribution, messaging, collaboration, and workflow; fine-grain and dynamic graphical composition of data representations; and a semantically rich infrastructure that facilities interoperability between components and the integration of external applications.

## 2.1 Component Model Base Requirements

An incremental approach to MCT component model development enabled the strategic realization of capabilities. For the first iteration implementation, it was highly desirable to postpone some design decisions until further programming experience was gained. Therefore, the initial set of requirements that are shared below was selected to build a component model where much of the interaction between components and the manipulation of component state were unconstrained. Note that requirements were also prioritized with respect to demonstrating proof of progress. An application prototype was written using a base implementation of the component model. This experience provided insights into the difficulty of developing a component-based application in the absence of programming language mechanisms (type checking, static interfaces, etc.) known to improve software robustness and reliability.

The following requirements drove the first iteration implementation of the component model:

- Reuse of components in alternate contexts (composability)
- Dynamic addition of attributes and behaviors
- First-class annotation of component data
- Loosely coupled interaction
- API domain independence
- Reaction to component state change
- Lifecycle management

## 2.2 Component Model Design, First Iteration

An introduction to the MCT first iteration component model design begins with its static model shown in Figure 1. Though the model contains names for some of our true Java interfaces and classes, this model should be read as a hybrid conceptual and implementation model as many of the classes shown have no corresponding source code class. Note that interesting features of the component model are intentionally left for description in another paper. The objective here is to provide the reader with enough background information to understand how degrees of agency are engineered into the component model.

At the heart of the MCT system is the concept of a reusable component being used as a foundational block from which applications can be built to interoperate. The fundamental capabilities of a component are defined in a set of interfaces that include IIdentity, IMessageable, and IComponentInternals, all of which are inherited by IComponent. Descriptions for each of these interfaces are provided below:

The non-inherited methods of the IComponent define a general-purpose API that establishes an interface for a general-purpose data structure that all components must support. Fundamentally, this data structure must be able to store component attributes and behaviors. Four of the primary requirements satisfied include the ability for a component to be composed of inheritable parts (computed value) or other components (served value), having state (stored value), and having the ability of a component to support the dynamic addition of behavior. These requirements are driven by the system's need to support a rich but general information model and to support the level of flexibility and extensibility needed to achieve the fine-grain composition of application components.

The general-purpose data structure API necessarily introduces an additional layer of abstraction above the Java programming language to circumvent the static type checking provided by Java and also supports the dynamic addition of arbitrary but well encapsulated behaviors (functions) that implement the IActor interface. Lack of static type checking leads to a more loosely coupled and more easily extensible system because new data types can be introduced that are not necessarily part of an a priori defined type hierarchy. The IActor interface provides the API necessary to effectively extend Java to support methods as parameters and methods as executable values.

Also realized by the IComponent interface is the requirement of providing full support for annotations. This, and the other data and behavior storage requirements have resulted in an API that defines the notions of and the operations for an internal component data model that is organized into fields, facets, notes, and values.

As shown in Figure 1, a component has zero or more fields. Each field is named and has a set of values. Fields and their values allow a component to store and retrieve both behavior (IActors) and attributes. A field may have zero or more facets. Facets are a kind of annotation that has a name and associated values. Facets allow a field to be further discriminated. For example, a field named "color" might hold a string of value "red" with a color facet of name "enumeration" where the facet values are "red," "green," and "blue." In this example, and a common use for facets, is the storage of typing information for a given field. Annotations on the value set of a field are also permitted, are called notes, and also have names. Adding to the previous example, one could add to the value set containing one value ("red") a note named "opacity" with a value of "0.95."

A core capability needed to satisfy the high level design goal of building a system that supports a rich user experience is to have a well-integrated event listening mechanism to detect and respond to changes in the content of a component so these changes can be readily propagated to the user interface for rendering. Since the attributes and behavior of a component are stored in its fields, facets, values, and notes; it is the manipulation of these structures that must be continuously watched to detect component changes. When a component change of interest is detected, the code for handling the event must be executed. This code, like the other behaviors of a component, is encapsulated within an object implementing the IActor interface.

The behaviors responsible for responding to component content changes can be further classified by the kind of operations they respond to, when the behaviors are executed, and the kinds of structures (note, value, field, and facet) a behavior watches. Behaviors can execute inline – the behaviors associated with a given data structure and operation type are executed before the operation that triggered its execution. This class of behaviors is modeled concretely in Figure 1 and includes CreationAccessor, GetAccessor, and PutAccessor. Behavior can also be executed asynchronously – the behaviors associated with a given data structure and operation type are added to a task queue and the operation is immediately completed. The task queue manages behavior execution including setting scheduling priority and the collapsing of behaviors to improve system performance while still maintaining semantic integrity. Note that inline behaviors can affect the results of an operation while asynchronous behaviors cannot. This class of behaviors descends from Listener, with the specializations ValueListener and NoteListener.

The IMessageable interface defines the general-purpose mechanism that is used for two components to communicate. A quick look at the API definitions reveals that the method signatures are very general; the passing of "messages" in MCT means communicating through the IMessageable interface via the exchange of Java Objects, the passing of a message type parameter that serves as a high-level indication of the kind of message that is being sent, and a return "message" that is also a Java Object. This is in contrast to traditional object-based

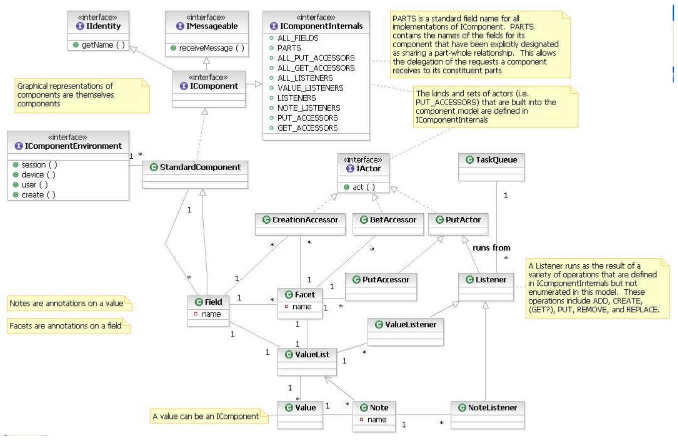
«interface»
IIdentity
getName ( )
«interface»
IMessageable
receiveMessage ( )
«interface»
IComponentInternals
ALL_FIELDS
PARTS
ALL_PUT_ACCESSORS
ALL_GET_ACCESSORS
ALL_LISTENERS
VALUE_LISTENERS
LISTENERS
NOTE_LISTENERS
PUT_ACCESSORS
GET_ACCESSORS
PARTS is a standard field name for all implementations of IComponent. PARTS contains the names of the fields for its component that have been explicitly designated as sharing a part-whole relationship. This allows the delegation of the requests a component receives to its constituent parts
«interface»
IComponent
Graphical representations of components are themselves components
The kinds and sets of actors (i.e. PUT_ACCESSORS) that are built into the component model are defined in IComponentInternals
«interface»
IComponentEnvironment
session ( )
device ( )
user ( )
create ( )
StandardComponent
«interface»
IActor
act ( )
TaskQueue
CreationAccessor
GetAccessor
PutActor
runs from
PutAccessor
Listener
A Listener runs as the result of a variety of operations that are defined in IComponentInternals but not enumerated in this model. These operations include ADD, CREATE, (GET?), PUT, REMOVE, and REPLACE.
Notes are annotations on a value
Facets are annotations on a field
Field
name
Facet
name
ValueListener
ValueList
Value
Note
name
NoteListener
A value can be an IComponent

**Figure 1 MCT component model, first iteration.**

systems that expose a set of methods that are associated with programmer inferred semantics, where their meaning comes from descriptive and distinctive method names, methods that are parameterized with a widely varying set of object types, and whose return types are frequently specialized. API's that are designed in this fashion blend into the object interface specifications application domain concepts. An alternate approach, one that is used heavily by the software agent community, is to remove the semantic meaning found in method signatures, define these concepts in a set of ontologies, and then use these ontologies to communicate intention within the body of the messages that are exchanged between agents. This is the solution offered by the IMessageable interface. This interface keeps components loosely coupled by providing a standard mechanism through which components can interact but in a way that is sufficiently general to allow communication between components that never knew about each other which talk about things that might never have been previously discussed. This is in contrast to traditional object oriented systems where who you can talk to and what you can talk about are more tightly constrained.

## 2.3 Component Role Description

The notion of a role is used by the MCT component model to define a set of attributes and behaviors that a component can have. The role description mechanism was fairly primitive for the first iteration implementation of the MCT component model but is being expanded as described in Section 2.4.4. Previously a role was an XML-based description of a set of attribute and behavior names that were associated with a role name. No further semantics were declaratively captured. For a component to play a given role, it must have contained fields with names for all of the role's attributes and behaviors.

## 2.4 Component Model Variability Points

A major goal of the second iteration design of the MCT component model is to introduce the language and system mechanisms necessary to achieve the C3I quality attributes and to generally provide developers with enough programming support to build robust components. Adding in these mechanisms as framework variability points was driven by the need for the component model to be sufficiently flexible to serve as the base for a wide range of mission operations software. This section presents these variability points as primitive language capabilities and API features. Additionally, as discussed in Section 3, these variability points are what make it possible for a component to take on customized degrees of agency.

MCT components execute within an infrastructure that is distributed and information-centric. The User Platform and Information Semantics Manager are two MCT subsystems that support the component model and its variability points. Each subsystem includes a variety of features, most of which are not shown in Figure 2. This figure enumerates a general use case sequence for the enforcement of component model configuration: (1) The User Platform is parameterized with a component model configuration and system introduced role definitions, (2) the User Platform delegates responsibility for maintaining configuration information to the Information Semantics Manager, (3) the application provides the platform with a description of application components, (4) the User Platform delegates the management of application component descriptions to the Information Semantics Manager, (5) the application requests component creation, (6) the User Platform requests a description of the component, (7) the component is instantiated, (8) the application uses the component, (9) the component base enforces the component model configuration.

All MCT information models including component configuration is ontologically expressed using the OWL language. The following sections discuss the variability points can be specified and enforced at runtime.

### *2.4.1 Data Typing*

Static versus dynamic type checking has long been a programming language design decision with an interesting trade space. The MCT component design attempts to circumvent dictating that a single type checking policy be enforced and instead opts for permitting no type checking, dynamic type checking, and pseudo-static type checking. The configured type checking policy of choice can be applied to constrain the messaging between components or the kinds of values that can be stored in the field of a component. Using the base implementation of the component model as-is, no type checking is performed. Supplying the system at runtime with an information model that richly describes the roles that a component can play permits the enforcement of dynamic type checking, where this type checking is coordinated with the Information Semantics Manager shown in Figure 2. Pseudo-static type checking is achieved by providing a component development tool with the information models and interpreting application code as it is written.

Performance, degree of coupling, reliability, rapid development, and component developer understanding are key considerations in selecting the data typing policy for a particular portion of an application.

Type checking has generally meant determining if a software

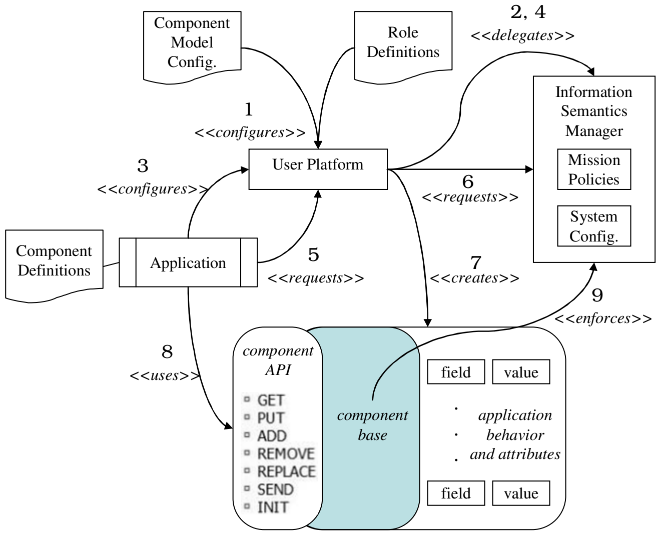

**Figure 2 Component model policy enforcement use case.**

element fits into a hierarchical categorization scheme. The functional specification of behavior enables a system to enforce type constraints based upon functional equivalence, which is a potentially more fine-grain typing mechanism that MCT is exploring.

### 2.4.2 Cardinality

The fields of an MCT component are multi-valued. Cardinality enforcement ensures that the number of values a field stores is within the bounds specified by the information model.

### 2.4.3 Field Names

Recall that field names are the symbolic ids associated with component attributes and behaviors. The base component model implementation permits the dynamic creation of fields where application code is free to choose their names. It is desirable to have system enforced field naming conventions that aid the intuitive understanding of component developers. An even stricter requirement is to only permit the use of field names that explicitly expressed in an information model. Both of these abilities are part of the second iteration component model design.

### 2.4.4 Field Semantics

The establishment of field semantics expressed using ontology-based information models is an important addition to the MCT component model because it enables components to reason about the behavior of other components, facilitates interoperability between components, provides system metadata that aids in the verification of component functionality, and adds a standards-defined layer to the MCT component model that improves its chances for wide scale adoption. Semantics can be associated with both the attributes and behaviors of a component. The design intention is to use portions of the prevailing semantic web services language as part of the MCT role description mechanism. Based on work in this area, one can anticipate varying levels of attribute and behavior specification including simple naming, parameter ordering, data typing, hierarchical behavior categorizations, and functional behavior description. Because the level of effort required to specify semantics and the degree to which these semantics can be used by the application vary considerably, the component model uses polices that dictate the amount of specification required to describe roles.

### 2.4.5 Component Malleability

The base implementation of the component model allows attributes and behaviors (fields) to be dynamically manipulated in any way seen fit including their removal. This makes the components very modifiable and permits a high degree of runtime adaptation. The downside to this flexibility is not being able to predict or rely on the presence of a particular attribute or behavior at any moment during program execution. Even though a role matching mechanism is provided by the Information Semantics Manager to determine if a component is playing a role, it is possible that the role could be modified directly after the test for satisfaction. Configuration of the component model permits varying degrees of modifiability that is categorized as open, immutable, additive, or replaceable. Open malleability allows attributes and behaviors to be freely modified while immutable malleability prohibits attribute or behavior changes once the component is instantiated via a declarative description or prototyped from an existing set of components. Additive malleability permits the addition of previously non-existent attributes or behaviors in contrast to replaceable malleability that prohibits the introduction of new attributes and behaviors but does allow existing ones to be overwritten.

### 2.4.6 Component Content Model Accessibility

The base implementation of the MCT component model circumvents the scoping mechanisms of Java in favor of publicly exposing the entire set of every component's attributes and behaviors. The MCT component model design hopes to re-introduce a richer form of scoping to regulate the visibility of component internals based on many considerations including user permissions, role-based permissions, and role type.

### 2.4.7 Time Alarm Access

The User Platform provides an alarm service that can be used, among other ways, to generate heartbeat events for components wishing to more proactively execute. Component model configuration permits the restriction of this service as appropriate for an application.

## 3. DEGREE OF AGENCY

A reasonable level of agreement on the definition of an agent must be in place before highlighting how variable degrees of agency are provided by the MCT component model. Starting first with an enumeration of their possible characteristics adapted from Franklin's classification properties [5], an implemented software agent can be communicative, proactive, reactive, autonomous, goal-directed, temporally continuous, mobile, learning, and intelligent. The component model is compared against these attributes in assessing degrees of agency, however, it is noted that the essence of agency is more narrowly defined by Wooldridge and Jennings [11, 12]. Paraphrasing their definition, an agent is a hardware or software-based computer system that is autonomous, social, reactive, and proactive. Autonomy implies that an agent functions without depending on user control and is ultimately in control of its actions and internal state. Having social ability is the requirement that an agent can communicate with other agents using an agent communication language. An agent is reactive if it can perceive and respond to its environment. Finally, an agent is proactive if it exhibits goal-directed behavior and takes the initiative to pursue its goals. Considering the core traits identified by this definition is useful in assessing the degree to which a component fundamentally acts like an agent.

In the subsections that follow, an assessment is performed of the MCT component model and system infrastructure's ability to support agency and multi-agent systems. New portions of the MCT system design are briefly described in addition to connecting how the component model variability points support degrees of agency.

## 3.1 Communicative

Communicative ability is a permanent feature of MCT components and a basic infrastructure exists to facilitate their interaction including white and yellow page platform services. As with many agent systems, three types of messaging are possible, (1) synchronous, (2) asynchronous, (3) and synchronous with timeout. Because components have access to richly described behavior semantics via query of the Information Semantics Manager, it is generally unnecessary to include static metadata with the communication of a message to another component.

## 3.2 Proactive

The proactive characteristic of a component can be regulated by limiting access to the system time alarm. In terms of an agent executing behavior in the absence of stimuli (application events), there is nothing more to the infrastructure for permitting agents to be proactive than allowing them to be pulsed with time events. The proactive characteristic of a component is also easily accomplished by the MCT system infrastructure by providing an alarm timer. Reactive components do not have access to the alarm timer and are therefore only allowed to execute in response to externally generated events. Proactive components can prompt themselves to execution using the timer alarm but can still respond to external events. Finally, the proactive pace can be regulated by policy enforced minimum slices of time between alarms.

## 3.3 Reactive

A reactive agent can perceive and respond to its environment. MCT components are supported by an infrastructure where semantically rich information is pervasive and its "environment" is an aggregation of information services that are provided by the User Platform and Information Semantics Manager. For the perceiving portion of reactivity, the degree of access that a component has to information is regulated by mission-defined policies including any security enforcement. A component exposes its behaviors according to the discussion given in Section 3.4 and can be configured such that no responses can be initiated by its environment.

## 3.4 Autonomous Components

Autonomous agents do not directly expose their content model, communicate through a generic domain-independent channel using a general-purpose information format (at the API level), do not directly expose their behavior for invocation, and have the option to refuse behavior service requests. Model configuration allows MCT components to take on varying levels of autonomy. For maximum autonomy, the corresponding component equivalent would provide no visibility to its content model, would play the part of no application-specific roles, and would service all messages received through the general message receiver role behavior where the information passed was only primitively typed. From this extreme, it is possible to provide a degree of autonomy that depends on the extent that roles are played, the level of specificity of these role descriptions, the satisfaction of the general message receiver role, and the kind of access policies enforced for the content model of a component. A component can or cannot be imbued with a large degree of autonomy.

## 3.5 Goal-Directed

A goal directed agent is generally thought to understand high level objectives and has the ability to pursue these goals. Trying to pin down what actually constitutes a high level of objective is difficult but the general notion is that an agent understands its tasks at multiple levels of abstraction and this knowledge influences how goals are completed. Framework support for goal-directed agent behavior can take many forms [2, 6, 7, 9] that can also be directly used as the internal component programming model. Rule engine usage by MCT components is a particularly well-suited programming model because its information structure is closely related to the structure of information provided by the MCT infrastructure. However, the MCT component model provides only a thin layer of indirection to route messages (frequent behavior requests) to their handler and does not dictate that any particular programming model be used. Instead, a variability point is available to introduce programming models that are suitable for a specific component or a specific component developer. MCT intends to provide a fundamental set of internal component programming models with its system frameworks but these are still to be determined.

## 3.6 Temporally Continuous

According to Franklin [5], a temporally continuous agent continuously runs within a process or processes. This characteristic attempts to identify a kind of implementation autonomy that an agent might exhibit though there are many implementation options available that make the characteristic less usefully distinctive. Since MCT components can be fine or coarse grain, implementing a one to one mapping of process to component does not scale. A thread pooling idiom is being considered by MCT that was previously used to provide pseudo-continuous execution for a P2P distributed agent platform for the National Airspace System was prototyped [4]. Its variability points included the ability to control the number of true threads shared by a group of agents and the ability to assign agents to specific groups. It is speculated that a similar design is suitable for MCT components.

### 3.7 Mobile

Mobile agents migrate from host to host. The benefits of mobile agents have been enumerated [8], a common use case for migration involves the motivation of an agent to switch hosts to achieve some computational advantage. To date, no system requirements exist that would necessitate designing-in component mobility so this characteristic is not offered in the same spirit as MAS agent mobility.

### 3.8 Intelligent

Intelligence is frequently measured by the degree to which an agent is able to use reasoning to pursue its objectives. The degree of intelligence of agents varies wildly depending on the application domain and usage of the AOP. From the perspective of implementation pragmatics, an intelligent agent typically, but definitely not always, includes a rule engine for inference. MCT provides an infrastructure for utilizing information semantics that facilitates intelligence by encoding knowledge and making it conveniently available and consumable. The Information Semantics Manger offers semantically rich description of component behaviors, ontology-based information model serving, and a comprehensive suite of knowledge transformation services. Additionally, MCT intends to provide a component with different kinds of reasoning abilities as was discussed in Section 3.5. The utilization of the MCT information infrastructure (via usage and configuration) and the utilization of the component behavior models (via parameterization), will determine the amount of intelligence exhibited by a component.

### 3.9 Learning

A learning agent becomes more intelligent over time as a result of its past experiences. Learning is part of the long term vision of MCT that imagines components adapting to their live execution environment and being self-configurable. The initial information infrastructure being established by MCT facilitates learning by offering concrete semantics that can be reasoned over. The ability to learn itself, like extending a component with goal-directed ability, is a component model variability point whose set of useful implementations is also to be determined and dependent upon customer need.

## 4. CONCLUSION

The Mission Control Technologies Project at NASA Ames Research Center is developing component-based middleware with multi-agent like characteristics that must satisfy many competing quality attributes. The goal in engineering degrees of agency is to build into an application framework the points of variability (hooks) that enable a programmer to tune the degree of agency used through customizations of the agent abstraction. MCT components are provided with a semantically rich information infrastructure that further supports their agency. With the hypothesis stated that engineering large-scale multi-agent systems requires framework support for achieving degrees of agency, the reader is left with early estimates of key variability points with empirical evidence to follow as the MCT team completes its next iteration of development.

## 5. ACKNOWLEDGMENTS

Special thanks to Robert Filman for his work in defining MCT's initial version of the component model. Thanks to members of the MCT team including Jay Trimble, Joan Walton, Harry Saddler, Tom Dayton, Dmitriy Lyubimov, Alan Tomotsugu, Dennis Heher, and Mitch Ai-Chang.

## 6. REFERENCES

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison Wesley, 2003

[2] R.S. Cost, Y. Chen, T. Finin, Y. Labrou, Y. Peng, "Using Colored Petri nets for Conversation Modeling," Issues in Agent Communication, Lecture Notes in Artificial Intelligence, Vol. 1916, May 1999, p. 178-192.

[3] FIPA, "FIPA Abstract Architecture Specification,"SC00001, December 2002.

[4] S.P. Fonseca and R. Filman, Technologies for System Wide Information Management P2P prototype, 2004

[5] S. Franklin and A. Graesser, "Is It an Agent, or Just a Program? A Taxonomy for Autonomous Agents," International Workshop on Agents, Agent Theories, Architectures, and Languages (ATAL), August 1996, p. 21-35.

[6] M.L. Griss, S.P. Fonseca, D. Cowan, R. Kessler, "Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors," Third Intl. Workshop on Agent-Oriented Software Engineering (AAMAS), July 2002, p. 113-125.

[7] Jess Expert System Shell, http://herzberg.ca.sandia.gov/jess/docs/.

[8] D.B. Lange and M. Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison-Wesley, 1998.

[9] A.S. Rao and M.P. Georgeff, "BDI Agents: From Theory to Practice, " Proceedings of the First International Conference on Multiagent Systems (ICMAS-95), June 1995, p. 312-319.

[10] R. Waterman, D. Smith, P. McCraw. Command, Control, Communications, Information (C3I) Architecture Reference Book, White Paper 1: Applications, October 2005.

[11] M. Wooldridge, "Agents as a Rorshach Test: A Response to Franklin and Graesser," International Workshop on Agents, Agent Theories, Architectures, and Languages (ATAL), August 1996, p. 47-48.

[12] M. Wooldridge and N.R. Jennings, "Agent Theories, Architectures, and Languages: A Survey," Wooldridge and Jennings editors, Intelligent Agents, Springer-Verlag, 1995, p. 1-22.